ETH zürich

ETH Library

Multi-User Secure Deletion on Agnostic Cloud Storage

Report

Author(s):
Wegberg, Gregor; Ritzdorf, Hubert; Capkun, Srdjan

Publication date:
2017

Permanent link:
https://doi.org/10.3929/ethz-a-010862066

Rights / license:
In Copyright - Non-Commercial Use Permitted

This page was generated automatically upon download from the ETH Zurich Research Collection.
For more information, please consult the Terms of use.

Multi-User Secure Deletion on Agnostic Cloud Storage

Gregor Wegberg
Oneconsult AG
Thalwil, Switzerland

Hubert Ritzdorf, Srdjan Čapkun
Department of Computer Science
ETH Zurich, Switzerland

Abstract

Cloud storage provides a large amount of storage space, allows customers to reduce, if not completely remove, their own storage infrastructure and comes with a low price tag. Enterprise customers can use cloud storage to provide well-connected collaboration platforms for their employees. However, cloud storage typically does not fulfil all the necessary requirements. In particular, current cloud storage does not provide *secure deletion*, i.e. the ability to irrecoverably delete data. Secure deletion is required to protect customer privacy and conform with data retention regulations. To address this issue, we present a *shared file system* with data stored on cloud storage and the capability for secure data deletion by any collaborator. Our system relies on trusted clients, an correctly-storing cloud storage provider and a trusted certificate authority. It does not require any additional server or communication channels besides the client's access to the cloud storage. We study different threats to secure deletion and make corresponding design choices. Finally, we show that our design is practical by implementing and evaluating a proof of concept based on Google's Cloud Storage.

1 Introduction

Cloud based services are an emerging product in use by a range of organisations, communities and individuals. They allow customers to build and use IT infrastructure running on top of infrastructure maintained by a third party. Cloud storage, one of the common cloud services, allows outsourcing data storage to the cloud, enabling customers to have virtually infinite storage space and associated services (e.g. backups and versioning) for a low cost *without* the need to run their own servers, or even data centers. The increasing availability of fast internet connectivity around the world and the globally distributed cloud infrastructure of major cloud providers makes cloud storage a prime choice to build a data sharing and collaboration system. However, the cloud storage service is operated by a third party and in most cases sensitive data is stored on it. Therefore, customers must either trust the cloud provider and store the data in plaintext, or deploy a trusted system on top of the cloud infrastructure reducing the necessary trust in the cloud service provider.

In this work, we build a collaboration system on top of cloud storage services, allowing members of a group to collaborate on data with the ability to *securely delete* it. This system shall not lessen the positive aspects of cloud services, i.e. low cost and the ease of deployment and operation.

Secure data deletion is the process of deleting data in an irrecoverable way. On most computer systems data deletion is performed in an insecure manner, i.e. the deleted

data can be trivially recovered. Solutions exists for secure deletion in case data is stored on a local storage device [37]. In case of cloud storage, however, these solutions are infeasible as no physical access to the storage medium is available. The data might reside on multiple physical storage devices and be part of different file systems; it might be replicated among multiple magnetic drives for fast access and redundancy, might be cached on SSD drives or might be backed up on offline magnetic tape for long-term storage. These factors also make it difficult for a cloud provider to offer secure data deletion on the physical storage devices. Even if offered, customers might not be willing to trust the cloud provider to correctly perform secure data deletion. Overall, for sensitive data storage the data owner should be responsible to assure secure data deletion.

For example, many countries' data privacy laws and bylaws include the destruction of data upon request or after retaining it is no longer necessary [34, 12, 10, 6, 9]. Lawmakers assume data destruction makes it irrecoverable, i.e. they assume the use of secure data deletion. The necessity for secure data deletion in the areas of healthcare, military and for any company concerned with the disclosure of information is obvious.

In this paper we present an approach to build a system allowing a group of collaborators to share, work on and securely delete data stored on cloud storage. This system uses trusted user clients to perform the necessary protocols and algorithms. A honest cloud storage provider is used to store data in large quantities, although any centralised storage service could be used. To establish trust at least one certificate authority must be trusted. No additional servers, trusted entities and communication channels, aside from a connection between clients and the cloud, are necessary.

We will present a proof of concept of our approach and conduct empirical evaluation of it using Google's Cloud Storage. This proof of concept uses FUSE (Filesystem in Userspace) to implement the operating system's file system interface and provide users with the illusion of a local data drive. This implementation will show the feasibility of our approach.

Our contribution is the design, implementation and evaluation of *shared* cloud storage allowing *multiple users* to share, work on and *securely delete* data while only *trusting* each other and a single digital certificate.

Section 2 introduces secure data deletion in more detail, presents ways to establish a secret among a group and discusses public key infrastructures briefly. In Section 3 we present the system requirements we deem important for a data sharing system with secure data deletion. Section 4 discusses our solution design in detail and is followed by the proof of concept presented in Section 5. Related work is presented in Section 6 and we conclude in Section 8.

2 Background

This section introduces the reader to secure data deletion, presents ways to establish secret keys among a group and provides a brief introduction to public key infrastructures. These three topics lay the foundation for our secure deletion on shared storage system.

2.1 Secure Data Deletion

Secure data deletion is the process of removing data from a system in such a way that "[...] if an adversary that is given some manner of access to the system is not

able to recover the deleted data from the system." [37]. Most common file systems (e.g. NTFS, ext2, ext3, FAT32) mark data as deleted by setting a flag or dereference a pointer in their metadata structures. This makes the recovery of deleted data trivial with commonly available tools [47, 44] as long as the deleted data is not overwritten on the physical storage device.

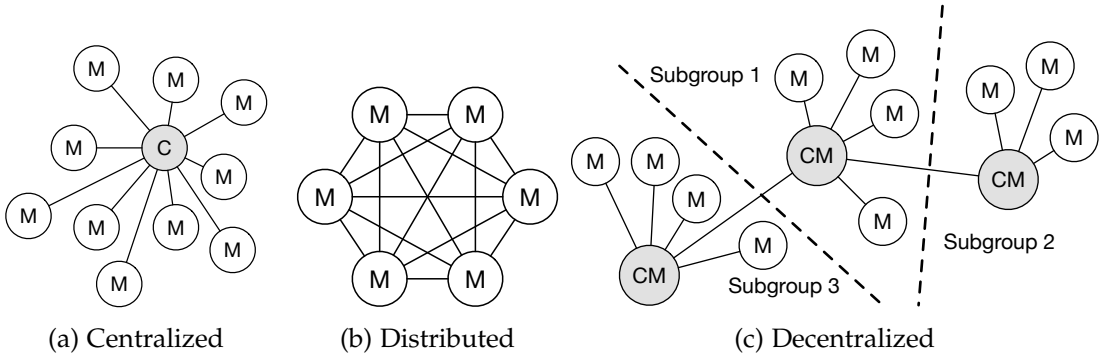
There are many ways to securely delete data from a storage device. They can be categorized along a multitude of dimensions; including the layer at which secure deletion is performed (e.g. user-level, file system, or device driver), the unit which can be securely deleted (e.g. file versus data block), the wear on the storage device and many other dimensions. Data can also be securely deleted by destroying the device that holds it through chemical or physical means (e.g. degaussing a magnetic drive). Such destructive solutions are out of the question for secure deletion of data stored on cloud storage. In the following we take a look at two basic non-destructive schemes employed to perform secure data deletion.

The first basic approach is to overwrite sensitive data with benign data. Although this approach is often recommended and employed [40, 27], it has multiple drawbacks and considerations to take into account. In case the attacker has access to a laboratory with the necessary instruments (e.g. a Magnetic Force Microscope or a Magnetic Force Scanning Tunnelling Microscope) they might be able to recover overwritten data [24]. Advancements in storage devices, like the use of flash memory or adding more advanced error handling, may allow an attacker to recover overwritten data. In case of flash memory based storage devices the physical characteristics prohibit the required functionality of in-place updates to perform secure deletion by overwriting data [37, 24].

The second approach is to use encryption. Before storing data on a storage device it is encrypted. To securely delete the data, it is enough to forget the key used for encryption. Usually such keys are also stored on a storage device and securely deleting the key results in securely deleting the encrypted data as no computationally bounded attacker can recover the key or access the data. The main goal of this approach is not to provide confidentiality, but should rather be thought of as a problem reduction where arbitrarily large data can be securely deleted by securely deleting a fix sized key [5]. This approach is very flexible. For instance, it allows the sensitive data to be stored on any storage device. Only the fixed size keys must be securely deleted. Due to the much smaller size of the keys compared to the sensitive data, it is possible to build cost effective small storage devices with secure deletion capabilities and use for the bulk of data large and cheap storage. Many different applications of this scheme were proposed and are summarized by Reardon et al. in [37].

To securely delete data stored on cloud storage services we employ the second approach, employing data encryption and the secure deletion of keys as a way to securely delete data stored in the cloud. Each unit of data (e.g. a file) is encrypted with its own random key, which can be forgotten, enabling us to provide secure deletion without any knowledge in regard to the storage devices holding the data. In addition, this approach can be parametrized based on the user's threat model (e.g. smaller/larger key sizes, nesting different encryption algorithms, etc.).

Adopting the overwrite scheme can be considered as impossible for data stored in the cloud. The owner of the data would need low-level access to identify all the locations where the data resides. It is likely replicated among multiple magnetic drives, backed up on magnetic tapes and cached by flash memory based storage. Such access is not provided by all major cloud storage providers and due to the complexity to provide this level of access will likely never be offered.



(a) Centralized (b) Distributed (c) Decentralized

Figure 1: Secure Multicast Communication protocol classes. "M" represents member nodes, "C" a centralized node performing key management tasks and "CM" are members chosen to perform key management tasks for their subgroup. Lines between nodes represent communication paths.

2.2 Group Key Establishment in Dynamic Groups

Our system provides a shared storage space allowing a group of people to collaborate and share data. To enable collaborators to access the data, which is encrypted to support secure data deletion (Section 2.1), all participants must have access to the encryption key. Hence, we require a way to establish a secret among the collaborators, a *group secret*, without outsiders having access to it.

This problem is similar to the research area of *Secure Multicast Communication*. Networks with *multicast* support allow a sender to address a data packet to a group of recipients, but send it just once. The network is responsible to deliver the data packet to all recipients. Although a multicast packet is addressed to a specific group, as with all (public) network communications we must assume it might be read by unauthorized network participants. Group members could also receive data packets sent by outsiders. Secure multicast communications, therefore, concerns itself with providing *confidentiality* and *authentication* for multicast based communication among a group. A very similar and overlapping research area is *Secure Group Communication*, which is not based on multicast capability of the underlying network. Many Secure Multicast Communication and Secure Group Communication protocols support dynamic groups where members may join and leave the group at any time. For our use case we are interested in this type of protocols providing the following security properties:

1. **Secrecy** A computationally bound attacker is not able to discover the group secret.

2. **Forward Secrecy** Passive attackers are unable to derive any future group secret from a subset of previously used group secrets. This is important in case a member leaves the group. A past group member must not be able to derive group secrets established after they have left the group.

3. **Backward Secrecy** With access to a subset of group secrets a passive attacker is unable to derive any of the previously used group secrets. This is important in case new members join the group. Any new member must not be able to derive group secrets used before it has joined the group.

As one might imagine, the moment a group secret is established securing the communication channel can be achieved with well-known techniques, i.e. symmetric encryption and message authentication codes. The main challenge and difference

between this kind of protocols is the way the group secret is established and distributed among the group members. These protocols can be classified into three broad classes [35]:

1. **Centralized** A central entity performs the group key management.

2. **Decentralized** A group is split into subgroups and each such group has a manager acting as the central key managing entity for that group.

3. **Distributed** Members generate and distribute the group secret among themselves. In most cases a subset of the members, or all group members, contribute to the group secret.

Figure 1 depicts all three cases. Nodes with an "M" are member nodes, "C" refers to a centralized node performing key management tasks and "CM" are member nodes chosen to perform key management tasks for their subgroup. Lines between nodes represent communication paths. In [35] S. Rafaeli and D. Hutchison provide an overview and comparison of such group secret management protocols.

Both, the *centralized* and *decentralized*, approaches assign key management and key distribution, in many cases also key generation, to at least one chosen member in the system. This member can either be part of the group or a dedicated node solely performing these tasks. Such an approach does not fit our purpose and requirements for multiple reasons:

1. Although the cloud storage service is a central system accessed by all members, we cannot use it for key management related tasks. Our model of the cloud storage service is a very simple storage device providing basic file operations and data persistence (Section 3.1). We cannot run applications on it.

2. Further, all group members are only allowed to communicate with the cloud storage service and there is no direct communication among group members (Section 3.1). Adding an additional special purpose node to the system would make the deployment and operation more complicated for organisations. For example, such a special purpose node would represent a single point of failure and be a very interesting target for an attacker. These problems can be managed, however, we aim for a system consisting only of member devices and a central cloud storage service (Section 3.1).

3. Lastly, members might disconnect from the network and with it disappear. In case one of the members chosen to perform key management would go offline the whole system might collapse. Solutions for this kind of problems exists (e.g. handing over the work to another node), but make the whole system more complex.

A *distributed* approach, however, meets all but one of our requirements. Any computation is performed by the participating members and they only need a communication network among them to exchange information in regard to the group secret. As we mentioned before, we do not allow clients to communicate directly with each other. However, this does not stop us from using this approach. We can use cloud storage for communication among members in regard to the group secret. Instead of providing communication paths between all members (Figure 1b) they communicate with the cloud storage service only (Figure 2). In addition, some distributed secure multicast communication protocols generate the group secret based on contributions

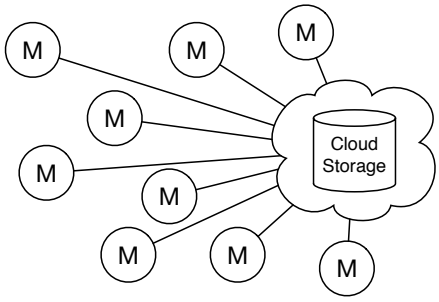


Figure 2: Distributed multicast communication protocol structure using cloud storage for communication between members.

from all members. This key agreement approach is more resilient to many types of attacks (e.g. no subset of the group can dictate the group secret) and usually uses a one-way function to compute the group secret from the contributions of all members.

3 System & Threat Model

This section presents our requirements consisting of the System Model and Threat Model. The System Model introduces the available components, interactions and communication channels. The Threat Model discusses the expected threats, threat actors and attacks.

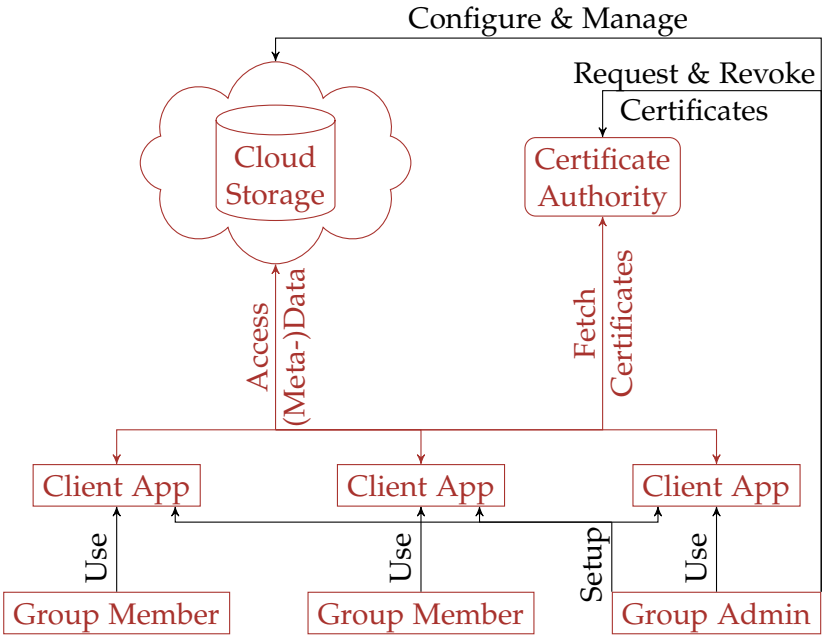


Figure 3: System model with the available components, interactions between actors and components, and the available communication channels between components. Red elements can be attacked by at least one of the presented attackers.

3.1 System Model

Figure 3 presents our system model. The system provides a multi-user cloud repository solution with secure deletion guaranteed by client-side operations. There are three actors: *Group Member*, *Group Administrator* and *Attacker*. The group members access the shared storage through their respective *Client App* and fully trust each other. The group administrator is a special group member in charge of the system configuration, including managing member certificates, configuring the cloud storage and setting up the client apps. For the attacker we present different models below that differ in their capabilities.

The actors can interact with the three components (*Client App*, *Cloud Storage* and *Certificate Authority*) as indicated by the lines connecting them. Note, that the client apps cannot directly communicate with each other as they use the cloud storage as sole communication medium. The cloud storage is completely agnostic to these operations. The cloud storage is only required to provide the standard file access operations, i.e., create, read, write, delete and is only trusted to correctly store data. Therefore, our design can make use of all major cloud storage providers. Additionally, we do not require any computational resources in the cloud, as all computation is carried out by the client apps, resulting in a completely server-less, client-based solution. Additionally, this limits the assumptions for our system model and makes our design more deployable.

To further improve the deployability, members should be able to dynamically leave and join. Given such a distributed with dynamic membership, the different client applications need a mechanism to verify each others contributions to the cloud storage. Therefore, we use a certificate authority (CA) that generates client certificates for all the clients and whose CA certificate is known to all client applications. The CA might be managed by the group admin or might be a third-party service.

System Assumptions Each group member uses its client app to interact with the system. The client app is assumed to be correct. We further assume that each client app has local access to a small storage medium on which it can perform secure deletion. We will discuss the security implications of the clients' storage media and eventual failures later. We assume no fixed availability for the client apps. In particular, client apps might go offline at any point and remain offline for an unbounded amount of time. Besides the correct storage of information no assumptions are made on the behaviour of the cloud storage. The cloud storage might keep data the user marked for deletion, e.g. on backups, and might share any stored or deleted data with the attacker. Additionally, the cloud storage might add unauthorized data to the storage. For the communication channels between client apps, CA and cloud storage we assume the Dolev-Yao model. Finally, we assume correct behaviour of the CA, in particular we assume non-forgable certificates and proper protection of the CA's private key.

3.2 Threats & Security Goals

Our security goals are derived from the attacker-centric threat model discussed in this section. The attackers presented here must be stopped by the system through built-in security mechanisms. However, before we introduce them it is important to identify the assets we aim to protect.

The system's primary goal is to provide secure data deletion for the shared cloud

storage. However, it is impossible to provide secure deletion for data which was at some point accessible by the attacker. If the attacker compromises a group member along with its client app, the attacker can by design access all currently available data on the cloud storage. Therefore, our goal is to provide secure deletion for all files during their lifetime no client app was ever compromised. This is due to the usability constraints the most we can possibly achieve in terms of secure deletion.

We now present different attacker models. All attackers are computationally bounded, have access to the complete system specification, however attacks are not persistent. To model persistent attacks the following attacks are executed after every system operation, repeatedly. Note, that Denial-of-Service (DoS) attacks are feasible through different mechanisms. However, they are outside the scope of our work.

Network Attacker A network adversary can eavesdrop and manipulate the communication between client apps, cloud storage and certificate authority. Therefore, for data transferred over these channels confidentiality and authentication have to be provided.

Misbehaving Cloud Third-party cloud storage system are a possible attacker. The *misbehaving cloud* attacker behaves honestly insofar as it does follow the agreed upon protocol, i.e. it stores and retrieves data correctly. However the cloud storage provider can aim to learn additional information and add unauthorized content to the storage. To protect against such an attacker, the system design has to provide confidentiality and authentication for data stored on the cloud storage. It is important to note that associated metadata, e.g. file names, also needs to be appropriately protected.

Subpoena Attacker The *subpoena attacker* can force cloud storage providers and group members, including the administrator, to fully disclose their knowledge. This includes data backups, saved private keys, memorized passwords and biometrical information. Therefore, the attacker has read access to all currently available plaintext data. Through the backups the attacker might also be able to retrieve all ciphertexts ever stored on the cloud storage. However, the attacker does *not* have access to data which was securely deleted by the client app. Note, that the subpoena attacker also models a powerful insider threat.

To protect against the subpoena attacker the client apps have to ensure proper secure deletion of locally available information and the system has to enforce secure deletion on the shared storage through local secure deletion by the client apps.

Faulty Certificate Authority The certificate authority might be misbehaving and as a result threaten the overall security. Although an attack on the CA is very problematic, it cannot be used by the attacker to access previously securely deleted data. However, it does put the correct functioning of the system including the secure deletion functionality at risk. Proactive security measures can be employed by the group and its group administrator. These include, the use of an intermediate certificate used for signing all member certificates and pinned by the client app.

In summary, the mitigations to be employed against attacks targeting the certificate authority, and the PKI in general, heavily depend on the exact threats the group faces and the attackers' capabilities.

M_i	The ith member in order of joining a group.
$\langle l, v \rangle$	The vth Key Tree node at height l. Both, v and l, usually refer to the position in the final tree.
$GS_k^{(i)}$	The kth group secret in epoch i.
$KTS_k^{(i)}$	The kth key tree secret in epoch i.
$DTS_k^{(i)}$	The kth data tree secret in epoch i.
$DTK_k^{(i)}$	The kth data tree key in epoch i.
$DHC_{(i)}$	Diffie-Hellman contribution by member M_i.
g	Exponentiation base used in Diffie-Hellman key exchanges.

Table 1: Notation used throughout this and following sections.

3.3 Non-Functional Goals

So far we have mentioned multiple times that we aim to set the bar low for groups to deploy, maintain and use such a data collaboration system. The ease of deployment and operation is non-functional requirement we consider. Similarly, we deem the usability of client apps as important.

As with many network based solutions, the performance of the system is of particular interest. In our case this refers primarily to the time spent accessing, storing and modifying data shared over the system among the group. Users expect to be able to create, open, save and delete data in a timely manner. This can be directly influenced by the design of the data structures, algorithms and, more generally, the implementation of the client app.

4 Methodology

The previous section presented our functional and non-functional requirements. In this section we propose an approach to meet these requirements.

Table 1 summarises the notation used throughout this and the following sections. Where necessary the notation is explained in more detail at first use.

4.1 Overview

In this Section we will present an outline our design, as seen in Figure 4. For more details please refer to [45]. The security, as described in Section 3.2, will be rooted in the procedures of the client apps. The client app is the only component of the system with usable, computational resources. To implement our secure data deletion scheme, as mentioned in Section 2.1, we leverage the securely-deleting medium attached to each client app, called *key storage*. To protect data confidentiality all group members need a shared secret. We use a *Key Tree* to derive the current *group secret* through multi-party Diffie-Hellman. Out of the group secret, members derive two secrets, the *Key Tree Secret* used in further levels of the key tree and the *Data Tree Secret* used to protect the confidentiality and authentication of stored data.

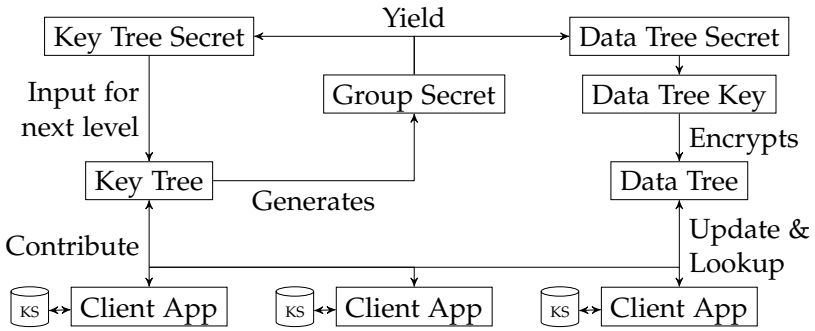


Figure 4: Solution Overview. A key tree is used to derive a common group secret out of which further secrets and keys are derived. KS is the securely-deleting key storage that is available to all client applications.

The data tree secret allows the derivation of the current *Data Key Tree* which encrypts the *Data Tree*. The data tree is responsible for two distinct tasks. First, it is used to represent a hierarchical file system, i.e. given a file path the tree can be traversed to reach the node associated with the provided path. The second task is to act as a tree of keys used to implement secure data deletion. Each parent node contains random symmetric keys to secure (confidentiality and integrity) its child nodes, and leaf nodes contain one random symmetric key to secure (confidentiality and integrity) the data associated with the leaf node. So far we have talked of sharing and securely deleting *data* without specifying the unit of data that can be shared and securely deleted. We have decided to share and securely delete *files*.

4.2 Secrets, Keys and Key Derivation

We use the term *key* for everything that is used as the key of a cryptographic function providing confidentiality, integrity or authenticity. Keys must confirm to the requirements of the cryptographic function and the security requirements of the group, e.g. in regard to the key size. Everything we refer to as a *secret* is solely used as *input key material* to *key derivation functions* and Diffie-Hellman key exchange. In other words, secrets are used to derive new secrets or keys.

The system requires the following secrets and keys. Their use will be discussed in detail throughout this section.

Group Secret This secret is established with help of the Key Tree (Section 4.4) and is the result of a repeated Diffie-Hellman key exchange. Group member contributes to this secret by providing a Diffie-Hellman contribution and all contributing members can compute the group secret. The group secret is used to derive all other keys and secrets.

Key Tree Secret The key tree secret is derived from the group secret and used as secret Diffie-Hellman contribution for the next level in the Key Tree (Section 4.4). We use $KTS_k^{(i)}$ as the notation for the kth key tree secret in the ith epoch (Section 4.5).

Data Tree Key To secure the Data Tree root node the data tree key is used (Section 4.3). The data tree key is derived from a data tree secret. $DTK_k^{(i)}$ is the kth data tree key in the ith epoch.

$$
\begin{tikzcd}
DHC_{(1)} \arrow[dr] & & & & \\
GS_0^{(i)} = g^{DHC_{(0)} \cdot DHC_{(1)}} \arrow[r, "F_1"] & KTS_0^{(i)} \arrow[r] & g^{KTS_0^{(i)} \cdot DHC_{(2)}} \arrow[r, "F_1"] & KTS_1^{(i)} \arrow[r, dotted] & {} \\
DHC_{(0)} \arrow[ur] & & DHC_{(2)} \arrow[ur] & &
\end{tikzcd}
$$

(a) Key tree secret chain. Members provide their Diffie-Hellman Contributions($DHC_{(i)}$) to derive the group secret and new key tree secret.

$$
\begin{tikzcd}
GS_0^{(i)} \arrow[r, "F_2"] & DTS_0^{(i)} \arrow[r, "F_4"] \arrow[d, "F_3"'] & DTS_1^{(i)} \arrow[r, "F_4"] \arrow[d, "F_3"'] & DTS_2^{(i)} \arrow[d, "F_3"'] \arrow[dr, dotted] & & & \\
& DTK_0^{(i)} \arrow[r, dotted] & DTK_1^{(i)} \arrow[r, dotted] & DTK_2^{(i)} \arrow[dr, dotted] & GS_1^{(i)} \arrow[r, "F_2"] & DTS_3^{(i)} \arrow[d, "F_3"'] \arrow[rr, dotted, "F_4 \circ \cdots \circ F_4"] \arrow[dd, dotted, "F_3 \cdots \cdots F_3"] & & DTS_k^{(i)} \arrow[d, "F_3"] \\
& & & & & DTK_3^{(i)} \arrow[rr, dotted] & & DTK_k^{(i)}
\end{tikzcd}
$$

(b) Derivation of data tree secret and data tree key chain based on the group secret.

Figure 5: Derivation of keys and secrets and their dependency.

Data Tree Secret Each time a new data tree key is necessary, first a *data tree secret* is derived. This secret is either derived from a group secret or the previous data tree secret. $DTS_k^{(i)}$ represents the kth data tree secret in the ith epoch.

4.2.1 Key Derivation and Key Dependencies

Figure 5 shows how the secrets and keys are connected with each other and derived from one another. Lightly dotted lines show the logical connection between secrets/keys, however no derivation is possible. For example, $DTS_3^{(i)}$ is not derived from $DTS_2^{(i)}$, but rather derived from $GS_1^{(i)}$. As the figure shows we require four key derivation functions, F_1, F_2, F_3 and F_4. Each of this key derivation functions is assumed to output a cryptographically strong pseudorandom key. Further, the key derivation functions are assumed to be one-way.

Each key tree secret is derived from a group secret as is shown in Figure 5a based on the Diffie-Hellman Contributions of different members. This secret is used as a Diffie-Hellman secret on the next level of the key tree to allow further join operations. The data tree secret and data tree key are closely related to each other, as is shown in Figure 5b. The data tree secret is either derived from a group secret, e.g. $GS_0^{(i)}$ or the previous data tree secret. Derivations from the previous data tree secret are very efficient as they require no new coordination between members. Data tree keys are always derived from the data tree secret with the same index. Each time the group changes, and a new group secret is established, a new data tree secret and data tree key must be derived. Otherwise, a new secret and key are only derived due to a change to the Data Tree.

Client store the current key tree secret, data tree secret and data tree key on their key storage. Once one of these keys is renewed previous key/secret is securely deleted. Otherwise, a subpoena attacker (Section 3.2) would be able to read the previous key/secret and use it to access data securely deleted before the attack.

4.2.2 Group Secret

The *group secret* is a secret only known to group members. As mentioned in Section 2.2 the group secret must satisfy three requirements: Secrecy, Forward Secrecy and Backward Secrecy. These properties are essential to support secure data deletion and meet the security requirements laid out in Section 3.2.

1. Without *Secrecy* an attacker can gain access to the group secret trivially.
2. Without *Forward Secrecy* an evicted member's knowledge (the previous group secret) can be used to calculate future secrets and with it access data shared and securely deleted in the future.
3. Without *Backward Secrecy* a group member's knowledge (access to the current group secret) can be leveraged to derive previous secrets and with it access previously securely deleted data.

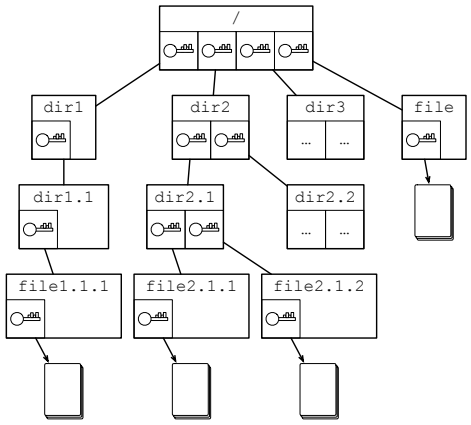
Further, we require the group secret to be established in a contributory fashion. Every group member must contribute an equal share to the group secret. This removes the possibility of a single point of failure; no single group member, or a subset of the group, can force a secret upon the whole group. Additionally, each group member contributes entropy to the resulting group secret. However, contributory key agreement protocols do not "[...] tolerate failures and group membership changes during execution" [26]. To resolve these shortcomings, we will use a key tree to update the group secret in response to group changes and use Diffie-Hellman key exchange to derive the group secret. Once the group secret is derived it is only used to derive further secrets and afterwards securely deleted from memory.

4.3 Data Tree

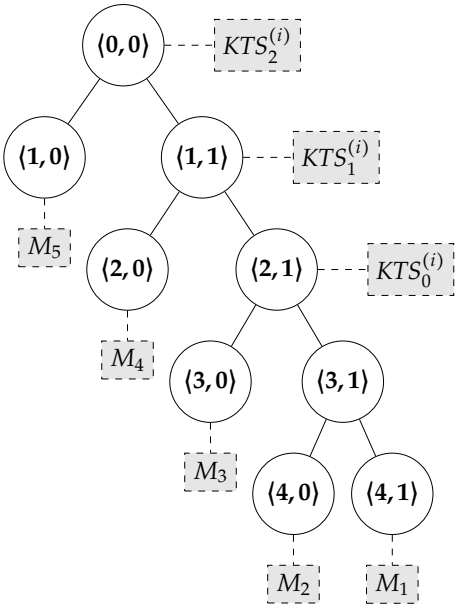
The Data Tree, as seen in Figure 6a, has manage the file hierarchy and enables efficient secure data deletion. To achieve this, every inner node of the Data Tree contains random symmetric keys for each of its children, providing confidentiality and integrity. The root node is secured the same way with the data tree key (Section 4.2). Data Tree leaf nodes hold one key and a reference to the file they represent. The file is *not* stored in plaintext, but we rather store the file encrypted under authenticated encryption using the key from the corresponding data tree leaf. To securely delete a file it is enough to remove its leaf node and refresh all keys along the path up to the root node, as explained in [38]. Such an operation triggers the derivation of a new data tree key and with it the derivation of a new data tee secret.

As the Data Tree describes the shared file system, it must be known to all group members. This is achieved by storing the whole tree on cloud storage. Members can access and cache parts of the tree locally to improve performance, however, any change must be committed to the Data Tree stored on cloud storage to be visible to all other group members. Note that metadata is protected as it is encrypted inside the data tree.

Attackers need the data tree key or keys from inner nodes to access any any data or metadata. However, the data tree key will be protected properly and without (one version of the) data tree key, computationally bounded attacker cannot derive keys from inner nodes. Due to the authenticated encryption, legitimate members can detect illegitimate modification to the data tree. The data tree is therefore protected from misbehaving cloud attackers.



(a) Exemplary Data Tree representing a file system with directories and files. Nodes contain symmetric, cryptographic keys to decrypt their child nodes. Leaf nodes contain the encryption keys.



(b) Example Key Tree with five members. The first three members formed the first group. M_4 and M_5 joined that group afterwards leading to two new group secrets and with it two additional key tree secrets.

4.4 Key Tree

We introduced the group secret and associated keys and secrets in Section 4.2. The group secret is the source of all other secrets and keys we require for our system. This secret is established with the help of the *Key Tree*, a binary tree containing the contributions of each participant. Figure 6b shows an example of a Key Tree where three members (M_1, M_2, M_3) joined and established the first group secret. Afterwards M_4 joined, leading to the establishment of a new group secret (a consequence of Backward Secrecy introduced in Section 2.2). The same applies for Member M_5. The gray, dashed boxes show logically associated data and objects. M_i refers to the ith group member in order of joining the Key Tree and $KTS_j^{(i)}$ represents a key tree secret associated with an inner node and marks nodes where a group secret was established.

The basic idea of the Key Tree is to perform repeated Diffie-Hellman key exchanges between a joining member and the existing group members. As with the Data Tree the Key Tree is readable and writable by every group member with access to the shared cloud storage. All algorithms and changes to the Key Tree are performed locally by client apps and committed afterwards to cloud storage. Our Key Tree approach is based on the work by Kim et al. in [25, 26]. In contrast to their approach we do not try to keep the key tree as balanced as possible, on the contrary, we require it to be unbalanced in order to make join operations as simple as possible.

4.5 Epochs

For a secure file deletion to be successful all keys and secrets, which can be used to access the securely deleted data, must be securely deleted. Every client app that participated and had access to the file before it was securely deleted must securely

delete all the secrets and keys that allowed it to access that file, only then the secure file deletion is successful. In Section 4.2.1 we already specified that a client app is only allowed to keep the secrets and keys it requires to participate at the current moment in time. All other secrets and keys, i.e. previously used ones, must always be securely deleted. In addition, we also recommend client apps to securely delete their whole key storage in case they are offline for some amount of time.

However, these approaches are enforced by the client app onto itself. To make sure the attack window and surface is as small as possible on securely deleted files we introduce *epochs* into the system. The basic idea is to divide the system's lifetime into epochs, each of which has its own Key Tree and as a consequence its own group secrets. The group secrets between two Key Trees are, by definition, independent. Group members must join an epoch's Key Tree to collaborate in that epoch. The switch from one epoch, for example epoch i, to another epoch, epoch $i+1$, is referred to as an *epoch transition*. Such an epoch transition is performed by one of the group members being part of both epochs' Key Tree. First, it calculates the first group secret of the next epoch's $(i+1)$ Key Tree, derives the keys and secrets and re-encrypt the Data Tree root node with the new data tree key. It is important to note that the first data tree secret and data tree key of epoch $i+1$ have no connection, aside from a logical one, to the last data tree secret and data tree key of epoch i (see Figure 5b in Section 4.2.1). The same is true for the key tree secret. This is a consequence of key independence of epoch i's group secrets and epoch $i+1$'s group secrets.

In addition, epochs allow group members to be informed when a previously executed secure deletion operation is successfully finished, i.e. no group member exists with secrets and keys that can be used to access that securely deleted data. As we have mentioned, each epoch has its own Key Tree. By keeping the old Key Trees and not deleting them at epoch transitions, a client app can derive the worst case content of every client app's key storage. The worst case is always the group secret created due to the group member joining the Key Tree and all the secrets and keys that can be derived from it. Calculating these worst case scenarios can also be of use as an auditing tool for the system.

5 Implementation

This section briefly introduces our implementation of the system presented in Section 4 and the evaluation thereof. For a more complete description, refer to [45].

5.1 Introduction

The major requirements discussed in Section 3 and components presented in Section 4 are implemented in the proof of concept we will discuss in this section. We employ Google's Cloud Storage service [19] as the cloud storage component. Google's Cloud Storage provides strong, global and instant consistency for *read-after-write*, *read-after-metadata-update* and *read-after-delete* operations [20] in contrast to other well known cloud storage services [15]. This simplifies the implementation of locks significantly and is the primary reason we chose Google's cloud storage service. Similar to all other major cloud storage services, it allows us to store *data objects* inside a *bucket*. Every data object can be given a unique name by which it is accessible at a later point in time. Data objects consist of data (the content) and metadata, both of which we can query and modify, except for a few metadata fields.

The *certificate authority* is our own and uses OpenSSL [32] to maintain the CA. However, the client app implementation is capable of using any X.509 certificate.

The *client app* is implemented in Python [13]. The client's *cryptography* module is based on the `cryptography` library [46] and is discussed in more detail in Section 5.3. The *user interface* implements a FUSE (Filesystem in Userspace [30]) drive and is explained in Section 5.5. To interact with Google Cloud Storage service we employ the `google-api-python-client` library [18, 22].

5.2 Encoding and Storage

All data is serialised to a JavaScript Object Notation (JSON) string and is stored inside a data object on cloud storage. This data format is human readable and with it makes it easier to verify data objects by hand and track down possible errors. In addition, Python's standard library includes a module to read and write JSON strings, reducing the need for additional dependencies. Although JSON is a good choice to build a proof of concept, a production ready implementation should likely use a more efficient binary format to reduce the time spend parsing and processing.

Each epoch's *Key Tree* is stored as a single serialised JSON string on cloud storage at a fixed location. In contrast, every *Data Tree* node is stored as its own data object. Only the root node is stored at a fixed location. All other nodes are stored under a name consisting of a fixed prefix (`/datatree/`) and a random suffix. The random suffix is a random UUID [29] (`00cd09ee-402e-4d76-9805-7a49f22332ad`). Using the node's name or the node's file or directory path would stop us from securely deleting all metadata and leak information.

A *file* shared in the system is first put through the necessary cryptographic functions and their output is stored as a JSON string inside a data object. Each file is stored in its own data object. Again, as with Data Tree's nodes, each files's data object name consists of a fixed prefix followed by a random UUID for the same reason as already presented.

5.3 Cryptography

The proposed approach requires a few cryptographic functions which are left to be selected by the implementation. In the following we will present our choices, their properties and provide our reasoning.

5.3.1 Authenticated Encryption

Data Tree nodes and associated files are secured with an authenticated encryption (Section 4.3). We use *AES-GCM* [11], i.e. AES in Galois/Counter Mode of operation, in these situations.

The AES keys are 256 bit long. This key length is considered to be secure up to the year 2030 [2] and is recommended for new systems [41]. Consequently, the key derivation function (F_3, Section 4.2.1) used to derive the data tree key must output a 256 bit long key.

The Initialisation Vector (IV) is 128 bit long and we store a 128 bit long tag for authenticity and integrity, hence, disallow tag truncation. It consists of 64 random bits, 32 bit representing the data tree key's index and 32 bit containing the least significant 32 bit of the current epoch number. This IV setup should assure a negligible likelihood of a repeating IV and follows recommendations presented in [11].

5.3.2 Diffie-Hellman

Diffie-Hellman key exchanges are performed over finite fields. The safe prime's length is 3072 bit long. According to [41, 3], this should be good enough for the near future.

5.3.3 Key Derivation Functions

The proof of concept does not use data tree secrets, but rather combines the data tree secret with the data tree key[1]. Therefore, the implementation does only require three derivation functions F_1, F_2 and F_3.

Based on the recommendation in [41] we use a key derivation function based on the Extraction-then-Expansion scheme. We selected the HMAC-based Extract-and-Expand Key Derivation Function (HKDF) [28] for all three derivations. Based on the recommendation in [8, 41] the following properties are set for all three key derivation functions:

1. The underlying *hash function* is *SHA-256*, i.e. both HKDF steps use HMAC-SHA-256.
2. Each key derivation uses a *salt* of the same length as the underlying hash function, i.e. 256 bit long random bit string.
3. The *info* field contains a *function identifier* to distinguish between the three key derivation functions, the current epoch number and the index of the key the HKDF will generate.

The *function identifier* allows us to create three different HKDF instances. It assumes the following values for each key derivation function we require:

1. `0x1` for F_1 deriving a key tree secret from a group secret.
2. `0x2` for F_2 deriving a data tree key from the previous data tree key.
3. `0x3` for F_3 deriving a data tree key from a group secret.

5.3.4 Public-Key Cryptography

Each member has a public and private key-pair. The public key is bound to the member's identity by a certificate signed by the certificate authority. Every key is 4096 bit long.

5.4 Synchronisation

To prevent race conditions and assure data integrity the presented implementation uses *readers-writer* locks; multiple client apps can always read data, but only one can hold a lock to perform a write operation. Our choice to use Google's Cloud Storage service is primarily based on its properties making the implementation of such locks easy (Section 5.1).

For read operations no locking mechanism must be implemented as Google's Cloud Storage behaves already in the expected manner. Any request to read a data object will be responded with the last successfully uploaded version of the requested data object.

[1] We refer to the key combining the data tree secret and data tree key as the *secure deletion key* in the implementation.

A writer lock is a data object stored on cloud storage with a well-defined name. For example, if the data object `/datatree/root` should be locked, a data object `/locks/ /datatree/root` is created. To make sure only one client app holds a lock we use the cloud service's precondition functionality [21]. With it we specify that a lock should only be created if no data object with the same name already exists. If one does already exist, an error is returned by the cloud storage, i.e. locking the data object failed.

5.5 User Interface

A group member interacts with its client app the same way as with any other file system the computer has access to. To do this the client uses FUSE (Filesystem in Userspace) [30] and the Python wrapper provided by `fusepy` [43]. The user does not see any obvious difference between the file system provided by its client app and any other file system, resulting in very good usability as users can use all the tools and applications they are familiar with and do not have to learn anything new. Deleting a file in the file system translates to a secure file deletion.

The proof of concept, however, requires right now the group member to launch it using a command line interface. This reduces the usability of the application. With additional engineering efforts this can be resolved.

The client is configured through a configuration file a group administrator can use.

5.6 Evaluation

This section presents and discusses some basic measurements of the proof of concept implementation. The data presented is intended to give a rough overview and show that the proposed approach is feasible.

5.6.1 Benchmark Setup

All measurements presented in this section are run on an unmodified Apple MacBook Pro, early 2013 edition, with the following specification:

- **CPU:** Intel® Core™ i7-3740QM CPU @ 2.70 GHz
- **RAM:** Two banks of 8 GB 1600 MHz DDR3 each
- **Operating System:** OS X 10.11.6
- **Kernel:** Darwin 15.6.0
- **Python:** Python 3.5.1
- **FUSE Driver:** FUSE for OS X (OSXFUSE) [14] 2.8.3

For benchmarking we use `fio` [1] in Version 2.13. It provides a wealth of options and a detailed statistical summary. All benchmarks configure `fio` as follows:

1. *sync* is used as the *ioengine*, i.e. `fio` invokes basic `read` and `write` system functions.
2. `fio` uses non-buffered I/O operations.
3. `fio` does not invoke the `fsync` system function.

Experiment	File Size	File Count	Total Runtime	Avg. Bandwidth (Read / Write)
exp7.1	2 MB	5	1487 µs	3445 kB s^{-1} / 3440 kB s^{-1}
exp7.2	2 MB	10	3123 µs	3262 kB s^{-1} / 3295 kB s^{-1}
exp7.3	2 MB	20	8298 µs	2458 kB s^{-1} / 2477 kB s^{-1}
exp7.4	2 MB	30	11 698 µs	2643 kB s^{-1} / 2608 kB s^{-1}
exp7.5	2 MB	100	38 253 µs	2677 kB s^{-1} / 2675 kB s^{-1}
exp7.6	2 MB	200	918 630 µs	221 kB s^{-1} / 224 kB s^{-1}
exp7.7	2 MB	300	406 736 µs	752 kB s^{-1} / 758 kB s^{-1}
exp8.1	128 kB	5	475 µs	690 kB s^{-1} / 656 kB s^{-1}
exp8.2	128 kB	10	864 µs	777 kB s^{-1} / 703 kB s^{-1}
exp8.3	128 kB	20	2584 µs	482 kB s^{-1} / 507 kB s^{-1}
exp8.4	128 kB	30	3243 µs	583 kB s^{-1} / 600 kB s^{-1}
exp8.5	128 kB	100	13 594 µs	471 kB s^{-1} / 469 kB s^{-1}
exp8.6	128 kB	200	170 472 µs	74 kB s^{-1} / 75 kB s^{-1}
exp8.7	128 kB	300	255 391 µs	74 kB s^{-1} / 75 kB s^{-1}

Table 2: Experimental results using Google's Cloud Storage FUSE client.

4. Only one thread is used.
5. At most 100 files are allowed to be open at the same time.
6. `fio` performs *mixed random reads and writes* (*randrw*).

The Google Cloud Storage instance used for all measurements uses one single *bucket*, "Standard Storage" Storage Class (see [23] for details) and is located in "europe-west-1" (see [16] for more information).

5.6.2 Reference Measurements

To have a reference to compare our proof of concept's performance against we use Google's Cloud Storage FUSE client [17] (`gcsfuse`). This application makes it possible to mount a Google Cloud Storage bucket as a file system.

Table 2 shows the results for 2 MB and 128 kB file sizes and different amount of files. This data is obtained using version `0.18.3` of the client, running on top of Go version `go1.6.3`.

5.6.3 Overall Performance

Table 3 shows an overview of the implementation's performance with different file sizes and amount of files. In these experiments two client apps are run in separate processes. One connects to the shared file system and mounts a FUSE drive with which `fio` interacts. We refer to this client as the "FUSE Client". A second client app ("Sponsor Client") is used to set up the system and support the first client in forming a group and to perform epoch transitions.

The first noticeable behaviour is that the bandwidth of the system dramatically drops in case the benchmark uses 200 or 300 files. The cause is due to limiting `fio` to at most 100 open files at the same time. The same drop can be observed in Table 2 for `gcsfuse`. Both clients, Google's and ours, upload changes to cloud storage the moment a file is closed [17]. By limiting the amount of open files we increase the number of data transferred to cloud storage greatly. For experiments with 100 or

Experiment	File Size	File Count	Total Runtime	Avg. Bandwidth (Read / Write)
exp2.1	2 MB	5	5857 µs	874 kB s^{-1} / 873 kB s^{-1}
exp2.2	2 MB	10	11 628 µs	876 kB s^{-1} / 885 kB s^{-1}
exp2.3	2 MB	20	23 146 µs	881 kB s^{-1} / 888 kB s^{-1}
exp2.4	2 MB	30	39 268 µs	787 kB s^{-1} / 777 kB s^{-1}
exp2.5	2 MB	100	123 273 µs	830 kB s^{-1} / 830 kB s^{-1}
exp2.6	2 MB	200	996 913 µs	204 kB s^{-1} / 206 kB s^{-1}
exp2.7	2 MB	300	1 263 512 µs	242 kB s^{-1} / 244 kB s^{-1}
exp6.1	1 MB	5	3857 µs	661 kB s^{-1} / 665 kB s^{-1}
exp6.2	1 MB	10	7996 µs	640 kB s^{-1} / 639 kB s^{-1}
exp6.3	1 MB	20	15 419 µs	660 kB s^{-1} / 667 kB s^{-1}
exp6.4	1 MB	30	23 714 µs	646 kB s^{-1} / 648 kB s^{-1}
exp6.5	1 MB	100	93 817 µs	544 kB s^{-1} / 546 kB s^{-1}
exp6.6	1 MB	200	810 267 µs	126 kB s^{-1} / 126 kB s^{-1}
exp6.7	1 MB	300	1 124 597 µs	135 kB s^{-1} / 137 kB s^{-1}
exp5.1	512 kB	5	3024 µs	412 kB s^{-1} / 433 kB s^{-1}
exp5.2	512 kB	10	5941 µs	429 kB s^{-1} / 432 kB s^{-1}
exp5.3	512 kB	20	11 344 µs	451 kB s^{-1} / 450 kB s^{-1}
exp5.4	512 kB	30	17 587 µs	437 kB s^{-1} / 435 kB s^{-1}
exp5.5	512 kB	100	57 359 µs	448 kB s^{-1} / 443 kB s^{-1}
exp5.6	512 kB	200	619 528 µs	82 kB s^{-1} / 82 kB s^{-1}
exp5.7	512 kB	300	1 043 322 µs	73 kB s^{-1} / 73 kB s^{-1}
exp4.1	128 kB	5	2145 µs	152 kB s^{-1} / 145 kB s^{-1}
exp4.2	128 kB	10	4444 µs	151 kB s^{-1} / 136 kB s^{-1}
exp4.3	128 kB	20	8626 µs	144 kB s^{-1} / 152 kB s^{-1}
exp4.4	128 kB	30	12 861 µs	147 kB s^{-1} / 151 kB s^{-1}
exp4.5	128 kB	100	44 311 µs	144 kB s^{-1} / 144 kB s^{-1}
exp4.6	128 kB	200	615 899 µs	20 kB s^{-1} / 20 kB s^{-1}
exp4.7	128 kB	300	1 034 439 µs	18 kB s^{-1} / 18 kB s^{-1}

Table 3: Overall performance for different file sizes and number of files in the root directory.

Exper.	Transitions	Avg. Execution Time FUSE Client (# calls) / Sponsor Client (# calls) `join_next_epoch`	`transition_to_next_epoch`
exp2.7	1	0.043 85 s (2×) / 0.059 28 s (1×)	0.038 26 s (25×) / 0.011 09 s (13×)
exp6.7	6	0.045 99 s (7×) / 0.057 19 s (5×)	0.115 28 s (20×) / 0.091 45 s (15×)
exp5.7	6	0.037 31 s (7×) / 0.046 50 s (5×)	0.106 81 s (17×) / 0.079 35 s (15×)
exp4.7	7	0.044 58 s (7×) / 0.049 18 s (6×)	0.129 31 s (18×) / 0.101 17 s (16×)

Table 4: Average time spent in epoch transition related methods of both clients, the FUSE Client and the Sponsor Client.

fewer files `fio` opens each file once, writes and reads to it, and finishes by closing it. In case of 200 or 300 files, however, `fio` opens files multiple times and closes them multiple times, and with it many more change commits to cloud storage.

A second observation is that the larger the file size the better the average bandwidth. The same behaviour can be observed in `gcsfuse` (Table 2). We assume this is due the host system's TCP *slow-start* behaviour, but requires additional investigation.

5.6.4 Epoch Transition

Table 4 shows the average time for one execution of the API methods `join_next_epoch` and `transition_to_next_epoch`, used to perform epoch transition related tasks. The former method lets the client join the next epoch, i.e. join the next epoch's Key Tree. The latter method tries to perform an epoch transition. However, such an epoch transition will only be executed if the next epoch's Key Tree has at least two members. This is the reason why `transition_to_next_epoch` might be called more times than epoch transitions are recorded. For Table 4 we selected the longest running experiments, i.e. the ones with the highest probability for multiple successful epoch transitions.

Experiment 2.7 (*exp2.7*) is interesting in so far that it is the longest running one, but could only perform one epoch transition. In this case, according to the logs, the Sponsor Client succeeded in joining epoch 1, but could not transition to it. The client failed to acquire the necessary locks and with it all calls to `transition_to_next_epoch` were unsuccessful. Consequently, this client stayed in epoch 1 and did not join the Key Tree for epoch 2. This lead to the FUSE Client not being able to transition into epoch 2 as the Key Tree for that epoch was not initialised yet.

The presented values clearly suggest that the overhead of epochs is negligible relative to the overall time. In fact, the worst case of all experiments with epoch transition enabled is *exp5.4* where the sum of all calls to `transition_to_next_epoch` take up 1.41 % (0.248 59 s) of the overall runtime (17 587 µs). In most experiments, however, it is clearly below 1 %.

6 Related Work

Secure deletion is a wide field of research with many proposed solutions. Reardon et al. [37] survey a number of approaches at different abstraction layers, systemize the properties, provide an adversarial model and compare the discussed solutions to each other. All presented solutions focus on a single computer interacting with local drives or with a commonly used file system. In our setting such approaches cannot be easily deployed, as discussed shortly in Section 4.1. However, they can be used to build the

member client's key storage.

Although proposals for secure deletion of data stored in the *cloud* exist, most of them require at least one additional entity that must be trusted. Xiong et al. [48] present an approach to securing data such that it can only be accessed during a specified time frame. After this time frame the data can be considered as securely deleted. However, the proposed scheme requires a trusted authority and time server. Tang et al. [42] develop an overlay for cloud storage services that provides secure data deletion. It too requires an additional trusted entity for key management and does not support collaboration among multiple entities. Further, only the creator of the data is able to securely delete it. Rahumed et al. present in [36] another approach, this time combining version control with secure data deletion for backups stored in the cloud. Again, an additional trusted entity is required for key management. This entity, however, can consist of multiple key-share holders. The approach proposed by Mo Z. [31] comes closest to our requirements. It only consists of trusted clients and the honest cloud storage. However, for multiple users to collaborate the *master key* must be shared by all users. The authors propose either a shared local storage or use of a local proxy server managing the necessary keys.

The research areas *Secure Multicast Communication* and *Secure Group Communication* provide a range of solutions to establish a secret key among a group. A great overview to start is presented by Rafaeli S. and Hutchison D. in [35]. Centralized approaches [7, 39, 33] require a central entity for key management which is against our requirements. Decentralized approaches, for instance [4], would require some of the participants to stay online all the time, or hand over their task of key management. This again does not meet our requirements.

Our key agreement is based on the distributed approach proposed by Kim Y. et al. in [25, 26]. Although other decentralized approaches exist, they mostly improve the cost of some operations by allowing asymmetric work assignment to nodes or by reducing the number of messages send over the network between two nodes. As we do not connect group members directly with each other, but rather use a central storage, most of these improvements are of no use to us.

7 Acknowledgements

This work was partly supported by the TREDISEC project (G.A. no 644412), funded by the European Union (EU) under the Information and Communication Technologies (ICT) theme of the Horizon 2020 (H2020) research and innovation programme.

8 Conclusion

In this paper, we set out to design, implement and evaluate a data collaboration system based on cloud storage with a secure data deletion capability. We presented our requirements based on a System and Attacker Model and proposed an approach to meet these requirements. By implementing and evaluating a proof of concept we have shown the feasibility of our proposed solution.

In contrast to the current literature our approach allows the implementation of a shared file system with secure data deletion, depending solely on trusted clients and a trusted certificate authority. It uses shared cloud storage and only assumes the cloud service provider to be honest and the availability of a communication channel between clients and the shared cloud storage. Neither additional channels between

the clients nor additional trusted entities are necessary. Although we assume the use of cloud storage throughout the paper, the same system can be run on any storage service accessible by all collaborators and providing very basic file I/O operations. The presented proof sketch and reasoning provide a basis to proof secure data deletion in our system.

Our proof of concept implementation shows that the proposed system can be implemented. Using FUSE as the user interface leads to a great increase in usability by giving the user the illusion of a local data drive. The empirical evaluation shows the basic implementation can already be used for collaboration. Shortcomings of the implemented client are identified, discussed and can be resolved with additional engineering efforts.

References

[1] Jens Axboe. *Fio – Flexible IO tester*. URL: `http://git.kernel.dk/?p=fio.git;a=summary` (visited on 2016-09-09).

[2] Elaine Barker. *Recommendation for Key Management Part 1: General*. Tech. rep. 2016-01. DOI: `10.6028/nist.sp.800-57pt1r4`.

[3] Elaine Barker et al. *Recommendation for Pair-Wise Key Establishment Schemes Using Discrete Logarithm Cryptography*. Tech. rep. 2013-05. DOI: `10.6028/nist.sp.800-56ar2`.

[4] Rakesh Bobba and Himanshu Khurana. "DLPKH – Distributed Logical Public-Key Hierarchy". In: *Information Systems Security: Third International Conference, ICISS 2007, Delhi, India, December 16-20, 2007. Proceedings*. Ed. by Patrick McDaniel and Shyam K. Gupta. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 110–127. ISBN: 978-3-540-77086-2. DOI: `10.1007/978-3-540-77086-2_9`.

[5] Dan Boneh and Richard J Lipton. "A Revocable Backup System." In: 1996, pp. 91–96.

[6] Bundestag. *Bundesdatenschutzgesetz*. URL: `https://dejure.org/gesetze/BDSG` (visited on 2016-09-12).

[7] Ran Canetti, Tal Malkin, and Kobbi Nissim. "Efficient Communication-Storage Tradeoffs for Multicast Encryption". In: *Advances in Cryptology — EUROCRYPT '99: International Conference on the Theory and Application of Cryptographic Techniques Prague, Czech Republic, May 2–6, 1999 Proceedings*. Ed. by Jacques Stern. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 459–474. ISBN: 978-3-540-48910-8. DOI: `10.1007/3-540-48910-X_32`.

[8] L Chen. *Recommendation for key derivation through extraction-then-expansion*. Tech. rep. 2011. DOI: `10.6028/nist.sp.800-56c`.

[9] Datenschutzbeauftrager des Kantons Zürich. *Merkblatt – Vernichtung elektronischer Daten*. URL: `https://dsb.zh.ch/internet/datenschutzbeauftragter/de/ueber_uns/formulare_und_merkblaetter.html` (visited on 2016-09-12).

[10] Die Bundesversammlung der Schweizerischen Eidgenossenschaft. *Bundesgesetz über den Datenschutz (DSG)*. URL: `https://www.admin.ch/opc/de/classified-compilation/19920153/index.html` (visited on 2016-09-12).

[11] M J Dworkin. *Recommendation for block cipher modes of operation: Galois/Counter Mode (GCM) and GMAC*. Tech. rep. 2007. DOI: `10.6028/nist.sp.800-38d`.

[12] European Parliament and of the Council. *Directive 95/46/EC of the European Parliament and of the Council of 24 October 1995 on the protection of individuals with regard to the processing of personal data and on the free movement of such data*. URL: http://eur-lex.europa.eu/legal-content/EN/TXT/?uri=CELEX%3A31995L0046 (visited on 2016-09-12).

[13] Python Software Foundation. *Python 3.5.2 documentation*. URL: https://docs.python.org/3.5/ (visited on 2016-09-08).

[14] *FUSE for OS X*. URL: https://osxfuse.github.io/ (visited on 2016-09-09).

[15] Andrew Gaul, Juan Manuel Combetto, and Jay Patel. *andrewgaul/are-we-consistent-yet: Eventual consistency of object stores*. URL: https://github.com/andrewgaul/are-we-consistent-yet (visited on 2016-09-08).

[16] Google. *Bucket Locations — Cloud Storage Documentation — Google Cloud Platform*. URL: https://cloud.google.com/storage/docs/bucket-locations (visited on 2016-09-15).

[17] Google. *Cloud Storage FUSE — Cloud Storage Documentation — Google Cloud Platform*. URL: https://cloud.google.com/storage/docs/gcs-fuse (visited on 2016-09-15).

[18] Google. *Cloud Storage JSON API Client Library for Python — API Client Library for Python — Google Developers*. URL: https://developers.google.com/api-client-library/python/apis/storage/v1 (visited on 2016-09-08).

[19] Google. *Cloud Storage - Online Data Storage — Google Cloud Platform*. URL: https://cloud.google.com/storage/.

[20] Google. *Consistency — Cloud Storage Documentation — Google Cloud Platform*. URL: https://cloud.google.com/storage/docs/consistency (visited on 2016-09-08).

[21] Google. *Generation Numbers and Preconditions - Preconditions*. URL: https://cloud.google.com/storage/docs/generations-preconditions#_Preconditions (visited on 2016-09-08).

[22] Google. *google/google-api-python-client*. URL: https://github.com/google/google-api-python-client/ (visited on 2016-09-08).

[23] Google. *Storage Classes — Cloud Storage Documentation — Google Cloud Platform*. URL: https://cloud.google.com/storage/docs/storage-classes (visited on 2016-09-15).

[24] Peter Gutmann. "Secure deletion of data from magnetic and solid-state memory". In: *Proceedings of the Sixth USENIX Security Symposium, San Jose, CA*. Vol. 14. 1996.

[25] Yongdae Kim, Adrian Perrig, and Gene Tsudik. "Simple and Fault-tolerant Key Agreement for Dynamic Collaborative Groups". In: *Proceedings of the 7th ACM Conference on Computer and Communications Security*. CCS '00. Athens, Greece: ACM, 2000, pp. 235–244. ISBN: 1-58113-203-4. DOI: 10.1145/352600.352638.

[26] Yongdae Kim, Adrian Perrig, and Gene Tsudik. "Tree-based Group Key Agreement". In: *ACM Trans. Inf. Syst. Secur.* 7.1 (2004-02), pp. 60–96. ISSN: 1094-9224. DOI: 10.1145/984334.984337.

[27] Richard Kissel et al. *Guidelines for Media Sanitization*. Tech. rep. 2014-12. DOI: 10.6028/nist.sp.800-88r1.

[28] Hugo Krawczyk. "Cryptographic extraction and key derivation: The HKDF scheme". In: *Advances in Cryptology–CRYPTO 2010*. Springer, 2010, pp. 631–648.

[29] P Leach, M Mealling, and R Salz. *RFC 4122: A Universally Unique Identifier (UUID) URN Namespace*. 2005. URL: https://www.ietf.org/rfc/rfc4122.txt (visited on 2016-09-08).

[30] *libfuse/libfuse: The reference implementation of the Linux FUSE (Filesystem in Userspace) interface*. URL: https://github.com/libfuse/libfuse (visited on 2016-09-08).

[31] Zhen Mo, Yan Qiao, and Shigang Chen. "Two-Party Fine-Grained Assured Deletion of Outsourced Data in Cloud Systems". In: *2014 IEEE 34th International Conference on Distributed Computing Systems*. Institute of Electrical & Electronics Engineers (IEEE), 2014-06. DOI: 10.1109/icdcs.2014.39.

[32] *OpenSSL - Cryptography and SSL/TLS Toolkit*. URL: https://www.openssl.org/ (visited on 2016-09-08).

[33] Adrian Perrig, Dawn Song, and J. D. Tygar. "ELK, a New Protocol for Efficient Large-Group Key Distribution". In: *Proceedings of the 2001 IEEE Symposium on Security and Privacy*. SP '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 247–. URL: http://dl.acm.org/citation.cfm?id=882495.884429.

[34] *Personal Information Protection and Electronic Documents Act*. URL: http://laws-lois.justice.gc.ca/eng/acts/p-8.6/page-1.html (visited on 2016-09-12).

[35] Sandro Rafaeli and David Hutchison. "A survey of key management for secure group communication". In: *CSUR* 35.3 (2003-09), pp. 309–329. DOI: 10.1145/937503.937506.

[36] Arthur Rahumed et al. "A Secure Cloud Backup System with Assured Deletion and Version Control". In: *2011 40th International Conference on Parallel Processing Workshops*. Institute of Electrical & Electronics Engineers (IEEE), 2011-09. DOI: 10.1109/icppw.2011.17.

[37] Joel Reardon, David Basin, and Srdjan Capkun. "Sok: Secure data deletion". In: *Security and Privacy (SP), 2013 IEEE Symposium on*. IEEE. 2013, pp. 301–315.

[38] Joel Reardon et al. "Secure data deletion from persistent media". In: *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM. 2013, pp. 271–284.

[39] A.T. Sherman and D.A. McGrew. "Key establishment in large dynamic groups using one-way function trees". In: *IEEE Transactions on Software Engineering* 29.5 (2003-05), pp. 444–458. DOI: 10.1109/tse.2003.1199073.

[40] Bundesamt für Sicherheit in der Informationstechnik. *IT-Grundschutz-Katalog – M 2.167 Auswahl geeigneter Verfahren zur Löschung oder Vernichtung von Daten*. URL: https://www.bsi.bund.de/DE/Themen/ITGrundschutz/ITGrundschutzKataloge/Inhalt/_content/m/m02/m02167.html (visited on 2016-08-27).

[41] Bundesamt für Sicherheit in der Informationstechnik. *Kryptographische Verfahren: Empfehlungen und Schlüssellängen. Teil 1*. Tech. rep. BSI TR-02102-1.

[42] Yang Tang et al. "Secure Overlay Cloud Storage with Access Control and Assured Deletion". In: *IEEE Transactions on Dependable and Secure Computing* 9.6 (2012-11), pp. 903–916. DOI: 10.1109/tdsc.2012.49.

[43] *terencehonles/fusepy: Simple ctypes bindings for FUSE*. URL: https://github.com/terencehonles/fusepy (visited on 2016-09-08).

[44] *Tools:Data Recovery — ForensicsWiki*. URL: http://www.forensicswiki.org/index.php?title=Tools:Data_Recovery&oldid=16222 (visited on 2016-09-09).

[45] Gregor Wegberg. "Client-Side Secure Deletion on Shared Cloud Storage". http://dx.doi.org/10.3929/ethz-a-010725398. MA thesis. ETH Zürich, 2016.

[46] *Welcome to cryptography*. URL: https://cryptography.io (visited on 2016-09-08).

[47] Wikipedia. *List of data recovery software — Wikipedia, The Free Encyclopedia*. URL: https://en.wikipedia.org/w/index.php?title=List_of_data_recovery_software&oldid=736218322 (visited on 2016-09-09).

[48] Jinbo Xiong et al. "A Secure Data Self-Destructing Scheme in Cloud Computing". In: *IEEE Transactions on Cloud Computing* 2.4 (2014-10), pp. 448–458. DOI: 10.1109/tcc.2014.2372758.